VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Algoritmy zpracování digitálního obrazu využívající technologii NVIDIA CUDA

Digital Image Processing Algorithms Utilizing NVIDIA CUDA Technology

2014 Jan Lesniak

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Jan Lesniak**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Algoritmy zpracování digitálního obrazu využívající technologii NVIDIA CUDA
Digital Image Processing Algorithms Utilizing NVIDIA CUDA Technology

Zásady pro vypracování:

Práce se bude zabývat akcelerací algoritmů pro zpracování obrazu pomocí technologie NVIDIA CUDA. Student musí nastudovat tuto technologii, porozumět ji a zpracovat několik vybraných grafických algoritmů (konvoluce, inverze, interpolace a další) pro CPU i GPU. Výsledkem bude srovnání obou kódů v oblasti složitosti implementace a použitých konstrukcí i rychlosti. Výsledná aplikace by měla mít uživatelské rozhraní v přijatelné formě pro možnost testování CPU i GPU implementací na uživatelem vybraných obrázcích.

Úkoly:
1. Nastudovat technologii NVIDIA CUDA a její základní vlasnosti.
2. Naprogramovat vybrané grafické algoritmy pro CPU i pro GPU. U GPU kódů, které mohou využít globální i sdílenou paměť, by student měl vyzkoušet oba přístupy.
3. Popsat technologii NVIDIA CUDA a její vlastnosti včetně použitých konstrukcí nutných k úspěšnému naprogramování tohoto algoritmu v prostředí CUDA (tedy zmínění problémů, se kterými se student při programování setkal a jejich řešení/vysvětlení).
4. Provést měření rychlostí obou implementací (případně více GPU implementací) a prezentovat je ve vhodné formě.
5. Vyvodit závěry o užitečnosti NVIDIA CUDA, zmínit rozdíly v práci pro programátora.

Seznam doporučené odborné literatury:

[1] NVIDIA CUDA Programming Guide
[2] http://drdobbs.com/cpp/207200659

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Milan Šurkala**

Datum zadání: 01.09.2013

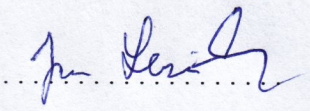
Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2014



Rád bych na tomto místě poděkoval panu Ing. Milanovi Šurkalovi za odborné vedení a cenné rady, které mi poskytoval během tvorby mé bakalářské práce.

Abstrakt

Tato práce se zabývá akcelerací algoritmů pro zpracování obrazu pomocí technologie NVIDIA CUDA a jejich porovnání s rychlostí zpracování běžným CPU. Popisuje a vysvětluje jednotlivé použité grafické algoritmy jak pro CPU, tak pro GPU s přístupy do globální a sdílené paměti. Výsledkem je srovnání kódů a jejich časů zpracování a vyvození závěrů o užitečnosti technologie NVIDIA CUDA.

Klíčová slova: NVIDIA CUDA, zpracování obrazu

Abstract

This thesis deals with image processing algorithms utilizing NVIDIA CUDA technology and comparison between processing speed of GPU and CPU. There is the description and explanation of used graphics algorithms for CPU and GPU with access to global and shared memory. Comparison of codes and their speed and conclusion of usefulness of NVIDIA CUDA technology is the result of this thesis.

Keywords: NVIDIA CUDA, image processing

Seznam použitých zkratek a symbolů

BGR	–	Blue Green Red
CD	–	Compact Disk
CPU	–	Central Processing Unit
CRT	–	Cathode Ray Tube
CUDA	–	Compute Unified Device Architecture
GPU	–	Graphics Processing Unit
SIMD	–	Single Instruction Multiple Data

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

V dnešní době se hodně využívají multimediální aplikace náročné na výpočetní čas a k jejich bezproblémovému a plynulému používání je třeba rychlý hardware. Protože se stejným postupem zpracovávají podobná data, může se výpočet řešit parelelně. K tomu lze využít vícejádrová CPU, instrukce podporující SIMD operace nebo právě grafické karty, které mají svou architekturou přirozeně obrovský potenciál k paralelním výpočtům. Existuje několik technologií zabývajících se výpočty na grafických kartách, kde nejrozšířenější je standard OpenCL. V této práci se však budu zabývat řešením společnosti NVIDIA, která uvedla svou vlastní technologii CUDA.

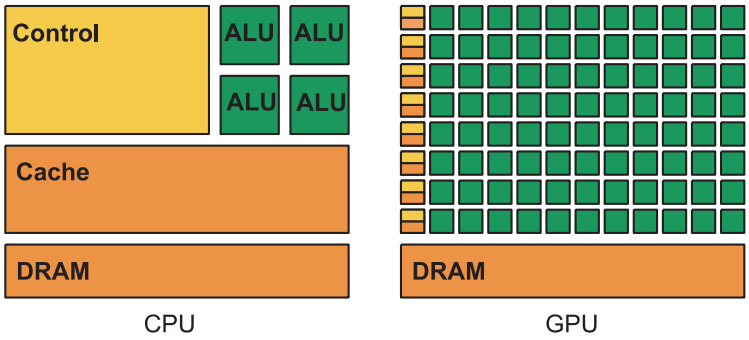
Ačkoli je napsání algoritmu pro platformu CUDA relativně snadné a stačí znát pouze několik málo novinek syntaxe či datových typů, k napsání dobře optimalizovaného kódu je však potřeba porozumět celé škále dalších věcí, které ovlivňují výpočetní čas. Ať už je to vyvarování se podmínkám v kódu, minimalizace přístupu do globální paměti nebo výběr správné matematické funkce, všechny tyto malé články, ze kterých se skládá výsledný algoritmus, mají nemalý vliv na výkon aplikace.

Práce je vytvořena jako program v jazyce C/C++ [1] s uživatelským rozhraním umožňující otevření jakéhokoli obrázku s následnou možností zvolení požadovaného algoritmu a jeho spuštění. Jako výsledek se objeví čas zpracování použitého algoritmu a obrázek s požadovanou úpravou. K vytvoření uživatelského rozhraní je použit vývojový program QT Creator [2]. Pro otevření a převedení obrázku do formátu BGR, kde je pro každou barvu použito 256 odstínů, je použita knihovna OpenCV [3]. K paralelnímu zpracování algoritmů pro CPU je použita knihovna direktiv OpenMP [4].

V kapitole 2 je platforma CUDA rozebrána podrobněji a jsou zde popsány hlavní principy fungování CUDA programů, jako uspořádání vláken, přístupy do pamětí a filozofie spuštění programu. V kapitole 3 jsou popsány použité algoritmy zpracování obrazu a vysvětleny jejich implementace jak pro CPU tak GPU s použitím globální i sdílené paměti grafické karty. V kapitole 4 jsou porovnány časy měření jednotlivých algoritmů v konfiguracích s paralelním a bez paralelního výpočtu pro CPU a se sdílenou nebo pouze s globální pamětí u GPU. Výsledky měření jsou shrnuty v poslední kapitole 5.

2 CUDA

CUDA je paralelní výpočetní architektura a programovací model vynalezen společností NVIDIA, který umožňuje dramaticky zvýšit výpočetní výkon pomocí grafických výpočetních jednotek. Byla zveřejněna v roce 2006 a prošla mnohými úpravami a zlepšeními a stále se vyvíjí. Využívá se v mnoha odvětvích, např. v astronomii, biologii, chemii, fyzice, analýzách a v dalších oborech [5], kde se mohou čerpat výhody paralelního zpracování dat. Jelikož základní myšlenkou paralelního zpracování je spuštění stejné části programu ve více vláknech, používá se v grafických čipech více tranzistorů ke zpracování dat, než ke cachování a řízení běhu, jak znázorňuje obrázek 1.



Obrázek 1: Porovnání počtu výpočetních jednotek mezi CPU a GPU [6]

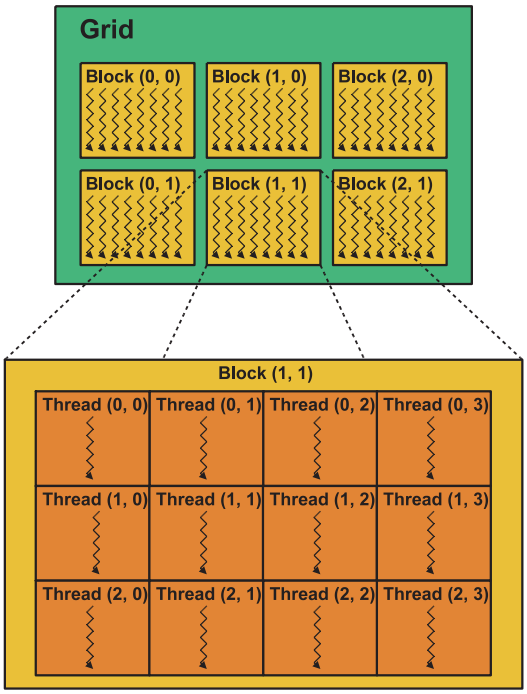
2.1 Uspořádání vláken

V programovacím modelu CUDA určuje programátor počet vláken, které se mají spustit. Vytváří skupiny vláken, které tvoří tzv. bloky a všechny bloky dohromady tvoří ucelenou tzv. mřížku. Jak bloky tak mřížka mohou být jedno, dvou nebo třídimenzionální. K identifikaci každého vlákna a bloku slouží jejich souřadnice. Na obrázku 2 můžeme vidět příklad dvoudimenzionální mřížky a bloků s vlákny. Programátor tedy definicí velikosti jednoho bloku a počtem bloků určuje počet všech vláken. Velikost bloku je však omezena maximálním počtem vláken v něm a počet bloků v mřížce je také omezen. Všechna tato omezení vychází z tzv. compute capability, tedy z výkonostní třídy, kam spadá určitá grafická karta. CUDA algoritmus se tímto může lišit v závislosti na cílové skupině grafických karet.

2.2 Typy pamětí

Každé vlákno může přistupovat k několika různým pamětem. Ty se odlišují svou velikostí, rychlostí přístupu a tím, co všechno k nim může přistupovat. Jejich typy jsou:

- Globální paměť, která je největší pamětí a má k ní přístup celá mřížka a hostitelský program, který tuto paměť využívá především k nahrání dat do grafické karty. Je sice pomalá, ale nyní již cachovaná.



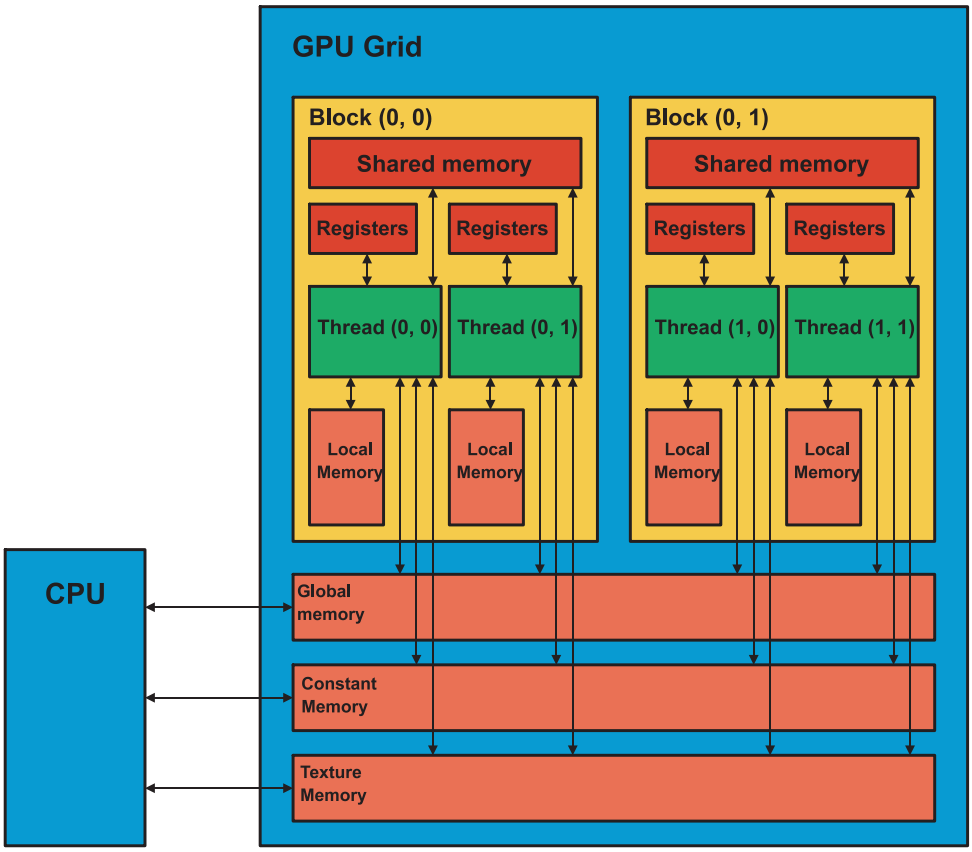
Obrázek 2: Mřížka složená z bloků a vláken [6]

- Sdílená paměť, která je společná pro každý blok vláken. Je velmi rychlá, ale aby využila svůj potenciál naplno, nesmí docházet ke konfliktu bank, ke kterému dochází, když více vláken současně požaduje data ze stejné banky.
- Registry, které slouží jako soukromá paměť pro každé vlákno a nikdo jiný k nim nemá přístup. Registry jsou nejrychlejší pamětí.
- Lokální paměť, která je stejně jako registry soukromá pro každé vlákno, ale je to vlastně pouze virtualizovaná globální paměť a ukládá se do ní vše, co se nemůže uložit do registrů.
- Paměť konstant, která slouží pro data, co se počas běhu vlákna nebudou měnit. Nachází se v globální paměti zařízení. Protože se cachuje, může být velmi rychlá, ale pokud cache chybí, je stejně pomalá jako globální paměť [7].

Obrázek 3 znázorňuje typy pamětí a přístupy k nim.

2.3 Běh programu

Typické spuštění CUDA programu probíhá tak, že hostitelský program zavolá funkci, která nakopíruje potřebná data do grafické karty, určí dimenzi mřížky a bloku, velikost bloku a jejich počet, případně určí velikost sdílené paměti a zavolá tzv. kernel, tedy hlavní



Obrázek 3: Přístupy k pamětem [8]

CUDA funkci, která je paralelizována vlákny. Musí jí být předány informace o mřížce a blocích a pak volitelně jakékoli předávané proměnné. Příklad volání kernelu můžeme vidět ve výpisu kódu 1, kde počet čísel definovaných v proměnné `blocks` určuje dimenzi a jejich hodnoty velikost mřížky v každé dimenzi. Podobně se v proměnné `threads` určuje počtem čísel dimenze bloku a jejich hodnoty určují velikost bloku. Toto volání kernelu tedy vytvoří dvoudimenzionální mřížku s 2×3 bloky a třídimenzionální bloky s $32 \times 16 \times 8$ vlákny a předá čtyři proměnné.

Poznámka 2.1 Datový typ `dim3` je struktura obsahující tři `unsigned int` proměnné pro každou dimenzi x, y a z. Přístup k těmto proměnným je možný tímto způsobem `proměnná.x proměnná.y proměnná.z` [9]. Ve výpisu kódu 1 jsou použity konstruktory pro nadefinování hodnot.

```
dim3 threads = (32, 16, 8);
dim3 blocks  = ( 2,  3);
kernel<<< blocks, threads >>>( A, B, C, D);
```

Výpis 1: Příklad volání kernelu

Po zavolání funkce kernel se tedy spustí příslušný počet vláken, která vykonají kód funkce a jsou ukončena. Při ukončení vlákna se uvolní veškerá paměť, kromě paměti globální. Výsledky, které vlákna zpracovala se tedy musí ukládat právě do této paměti před ukončením kernelu.

Pro identifikaci mřížek, bloků a vláken slouží speciální proměnné `gridDim`, `blockDim`, `blockIdx` a `threadIdx`. Proměnné `gridDim` a `blockDim` jsou typu `dim3`. První z nich `gridDim` určuje velikost mřížky a to pro každou z dimenzí. Podobná proměnná `blockDim` určuje velikost pro každou dimenzi bloku a proměnné `blockIdx` a `threadIdx` jsou typu `uint3` a nesou informaci o umístění vlákna, ve kterém bloku a kde v něm se nachází.

Poznámka 2.2 Datový typ `uint3` obsahuje stejně jako `dim3` tři `unsigned int` proměnné pro každou dimenzi x, y a z. Rozdíl je v tom, že `uint3` nemá konstruktor [9].

2.4 Průběh programu pro zpracování 2D obrazu

Ke zpracování obrazu je nutné předat obraz zpracovávající CUDA funkci v nějaké vhodné formě. Zvolil jsem BGR formát s 256 odstíny pro každou z těchto tří barev a obrázek je předán jako jednorozměrné pole, kde je každý záznam datového typu `uchar4` a představuje jeden pixel obrazu. Každý záznam nese informaci o jasu všech tří barev. Funkci na zpracování obrazu je předána velikost obrázku a také ukazatel na paměť pro výstupní obrázek.

Poté se alokuje paměť v grafické kartě pro vstupní a výsledný obrázek. Alokace je ukázána ve výpisu kódu 2. Slouží k ní funkce `cudaMalloc()`, kde parametry jsou adresa počátku obrázku a velikost alokované paměti. Proměnné `sizex` a `sizey` určují šířku a výšku obrázku. Návratová hodnota funkce určuje zdárnost alokování paměti a je potřeba tuto hodnotu testovat a při neúspěchu vypsat chybu.

```
cudaError_t cerr;
uchar4 *cudaInputPic;
uchar4 *cudaFinalPic;
cerr = cudaMalloc(&cudaInputPic, sizex*sizey*sizeof(uchar4));
if (cerr != cudaSuccess)
    printf ("1 CUDA Error [%d] - '%s'\n", __LINE__, cudaGetErrorString(cerr));
cerr = cudaMalloc(&cudaFinalPic, sizex*sizey*sizeof(uchar4));
if (cerr != cudaSuccess)
    printf ("2 CUDA Error [%d] - '%s'\n", __LINE__, cudaGetErrorString(cerr));
```

Výpis 2: Alokování paměti pro vstupní a výstupní obrázek

Po alokování paměti pro vstupní obrázek, je potřeba ho na toto místo překopírovat. K tomuto účelu slouží funkce `cudaMemcpy`. Parametry jsou popořadě ukazatel na vstupní obrázek, ukazatel do paměti grafické karty, kam chceme vstupní obrázek nakopírovat, jeho velikost a poslední parametr určuje směr kopírování. V tomto případě chceme z paměti počítače překopírovat data do paměti grafické karty. A opět je potřeba otestovat úspěšnost akce.

```
cerr = cudaMemcpy( cudaInputPic, inputPic, sizex * sizey * sizeof( uchar4 ),
      cudaMemcpyHostToDevice );
if ( cerr != cudaSuccess )
    printf ( "3 CUDA Error [%d] - '%s'\n", __LINE__, cudaGetErrorString( cerr ) );
```

Výpis 3: Překopírování vstupního obrázku do grafické karty

Dále je třeba určit velikost mřížky a bloků ke spuštění kernelů a paralelizovat tak zpracování. Každé vlákno zpracuje jeden pixel obrazu. Pro jednoduché adresování bez zbytečného přepočítávání indexů stačí dvoudimenzionální mřížka a bloky, kde poloha pixelu v obrázku bude korespondovat s polohou vlákna, které tento pixel zpracovává. Ve výpisu kódu 4 je zvolen čtvercový tvar bloku o velikosti 32 x 32 vláken, což je dohromady 1024 vláken, tvoří tak v tuto chvíli maximální velikost bloku a toto číslo je uloženo v proměnné `threads`. Proměnná `blocks` určuje počet bloků v mřížce, který je vypočítán z velikosti obrázku a velikosti bloku a pro každou dimenzi mřížky.

```
int block = 32;
dim3 threads( block, block );
dim3 blocks( ( sizex + block - 1 ) / block, ( sizey + block - 1 ) / block );
```

Výpis 4: Výpočet velikosti mřížky podle velikosti zpracovávaného obrázku

Když je paměť naalokována, data zkopírována a bloky s mřížkou nadefinovány, je možno spustit kernel a předat tak úlohu grafické kartě. Karta zpracuje obraz a po ukončení všech vláken se také ukončí i kernel. Výstupní obrázek se musí vykopírovat z globální paměti karty do paměti počítače. K tomuto účelu slouží opět funkce `cudaMemcpy`, ale již s parametrem `cudaMemcpyDeviceToHost`, který značí kopírování opačným směrem. Nakonec je potřeba uvolnit naalokovanou paměť jak pro vstupní tak výstupní obrázek, jak je vidět ve výpisu kódu 5.

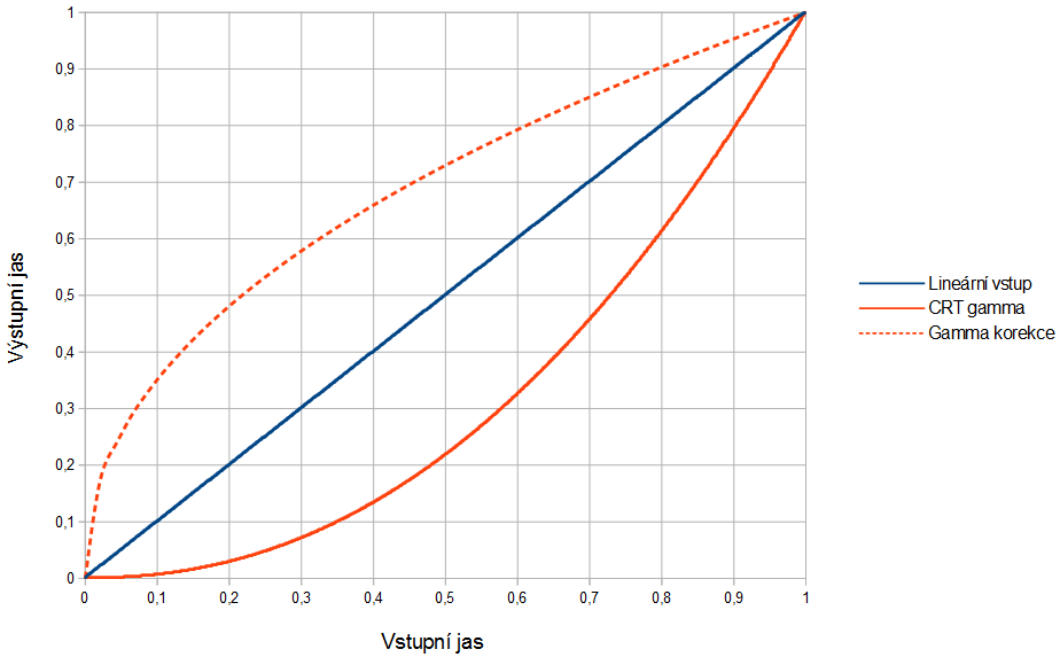
```
cudaFree( cudaInputPic );
cudaFree( cudaFinalPic );
```

Výpis 5: Uvolnění paměti v grafické kartě

3 Algoritmy ke zpracování obrazu

3.1 Gamma korekce

Gamma korekce se začala používat u televizních CRT obrazovek, kde je nelineární vztah v urychlovači elektronů mezi vstupním napětím a jasem paprsku. Průběh intenzity jasu přibližně odpovídá křivce $g = (f + \epsilon)^{\gamma}$, kde f jsou vstupní hodnoty jasu v rozsahu $f \in \langle 0, 1 \rangle$, ϵ je konstanta určující posun od počátku, ale většinou se zanedbává a γ je konstanta určující zkreslení jasu. Čím větší je hodnota γ, tím větší je rozdíl mezi vstupním a zobrazeným jasem. U většiny CRT monitorů se hodnota γ pohybuje v rozmezí od 1.8 do 2.4. Pokud má monitor úroveň γ rovnu například 2.2, je při vstupní jasu 0.5 výsledný jas 0.218, což znamená ztrátu více než poloviny jasu. Při použití převrácené hodnoty γ ve stejném vzorci, dostaneme křivku převrácenou podle diagonály, podle které opravíme výstupní intezitu jasu. Obě tyto křivky, kde je hodnota $\gamma = 2.2$ jsou na obrázku 4. Gamma korekce se také využívá při úpravě fotek, které vznikly špatnou expozicí, nebo při úpravě rentgenových snímků [10].



Obrázek 4: Graf průběhů intenzit jasů

3.1.1 Cuda program pro výpočet gamma korekce

Při volání kernelu pro gamma korekci si kromě vstupního a výstupního ukazatele na obrázek a jeho velikosti musíme předat gamma koeficient. Jak je zmíněno v kapitole 2.4, je vytvořená mřížka větší než obrázek a proto je třeba ukončit vlákna, která ho svým

indexem přesahují. Ve výpisu kódu 6 je vidět vypočítání indexu vlákna a jeho následné ukončení, pokud je za hranicí obrázku. Každé vlákno tak zná svou x a y pozici v obrázku, ale protože obrázek je jednorozměrné pole, je pozice přepočítána a výsledek je uložen v proměnné `position`. Proměnná `pom` představuje pixel vstupního obrázku pro vlákno, jehož pozice v mřížce odpovídá pozici v obrázku.

```
int y = blockDim.y * blockIdx.y + threadIdx.y;
if ( y >= sizey ) return;
int x = blockDim.x * blockIdx.x + threadIdx.x;
if ( x >= sizex ) return;

int position = y * sizex + x;
uchar4 pom = inputPic[position];
```

Výpis 6: Ukončení nepotřebných vláken

Nyní je vše připraveno pro výpočet gamma korekce. Protože se obraz skládá ze tří barevných složek, musí se gamma korekce vypočítat pro každou z nich. Proměnná `koef` je požadovaný gamma koeficient a protože chceme korekci, je jeho hodnota převrácena. Jelikož vstupní hodnoty f mohou podle vzorce nabývat hodnot od nuly po jedničku, musí se hodnoty jasu v programu podělit počtem odstínů každé barvy, aby tuto podmínku splňovaly. Po výpočtu je však potřeba stejnou hodnotou jas vynásobit, abychom dostali zpátky pro nás použitelné hodnoty. Může nastat, že hodnoty jasu budou větší než maximální hodnota jasu a to by znamenalo přetečení proměnné `final`, proto funkcí `__min()` jednoduše takové hodnoty ořežeme. Kvůli minimalizaci přístupu do globální paměti se výsledky ukládají do pomocné proměnné `final` a jakmile je proměnná naplněna, zkopíruje se její obsah do globální paměti, kam ukazuje `finalPic`. Takto se pixely vyskládá paralelně celý obrázek.

```
uchar4 final;
final[ position ].x = __min(255, pow(pom.x / (float)255, (1 / (float)koef)) * 255);
final[ position ].y = __min(255, pow(pom.y / (float)255, (1 / (float)koef)) * 255);
final[ position ].z = __min(255, pow(pom.z / (float)255, (1 / (float)koef)) * 255);

finalPic[ position ] = final;
```

Výpis 7: Výpočet gamma korekce

Tato úprava obrázku je nejjednoduší, protože se pouze načte samotný pixel, zpracuje se a jeho nová podoba se uloží do výsledného obrázku. Tento způsob lze použít pro další úpravy obrazu jako jsou inverze barev, převod na stupně šedi a různé barevné korekce.

3.1.2 Gamma korekce bez použití grafické karty

Zpracování obrazu procesorem s jedním jádrem probíhá výhradně sériově, tedy pixel po pixelu se postupně prochází celý obrázek. Současné procesory jsou ale už vícejádrové a tak je i potřeba této skutečnosti využívat a to především ve zpracování obrazu. V této práci jsem použil knihovnu direktiv OpenMP, která poměrně jednoduchým způsobem umí rozdělit výpočty v algoritmech do více jader procesoru. Ve výpisu kódu 8 jsou

dva cykly `for`, které slouží pro průchod pixely ve vstupním obrazu. Nad těmito cykly se nachází direktivy knihovny OpenMP, které slouží k tomu, aby rozdělily cyklus `for` na několik části, kde každou část bude zpracovávat jedno jádro procesoru. Celý tento kód stačí k tomu, abychom provedli gamma korekci. Liší se od CUDA kódu kernelu pouze v tom, že obsahuje navíc cykly pro procházení vstupních pixelů, ale na druhé straně odpadá práce s identifikováním vláken pomocí indexů v mřížce a blocích. Protože je u dalších CUDA algoritmů pro globální paměť grafické karty rozdíl pouze v tomto ohledu a kód je stejný, nebo minimálně velmi podobný, další popisování kódu pro CPU by bylo zbytečným.

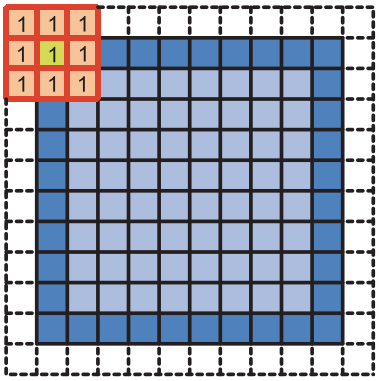
```
#pragma omp parallel
        {
#pragma omp for
            for ( int y = 0; y < imgSizeY; y++ )
            {
                for ( int x  = 0; x < imgSizeX; x++ )
                {
                    position = y * imgSizeX + x;
                    outputPic[ position ].x = __min(255, pow(inputPic[position].x / (float)255, (1
                        / (float)koef)) * 255);
                    outputPic[ position ].y = __min(255, pow(inputPic[position].y / (float)255, (1
                        / (float)koef)) * 255);
                    outputPic[ position ].z = __min(255, pow(inputPic[position].z / (float)255, (1
                        / (float)koef)) * 255);
                }

            }
        }
```

Výpis 8: Algoritmus výpočtu gamma korekce s použitím knihovny openMP

3.2 Rozmazání obrazu

Oproti předchozímu algoritmu se k úpravě obrazu využívá nejen aktuálně zpracovávaný pixel, ale i jeho okolí, tedy sousední pixely. Všechny tyto pixely jsou zpracovány podle masky, která představuje matici čísel. Matice je tvaru čtverce a délku strany má lichou, aby se její střed skládal pouze z jednoho pixelu. Číslo z matice je přiřazeno jednomu pixelu v obrázku a udává tomuto pixelu váhu, kterou je pak zpracovávaný středový pixel ovlivněn.

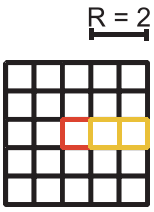
Nejjednoduším algoritmem je rozmazání obrazu pomocí matice složené ze samých jedniček. Jedná se de facto o zprůměrování jasu pixelů pokrytých maticí [10]. Jas pixelu se vynásobí příslušným číslem v matici, v tomto případě jedničkou, pak jsou všechny jasy sečteny a následně poděleny počtem pixelů pokrytých maticí. Příklad masky 3×3 pixely je znázorněn na obrázku 5. Maska obsahuje pouze jedničky a zpracovává pixel s indexem (0, 0). Pixelům, které jsou za hranicí obrázku, jsou přiřazeny hodnoty jasu z nejkrajnějších pixelů, které jsou označeny tmavě modrou barvu.



Obrázek 5: Použití masky na obrázek

3.2.1 Rozmazání obrazu s použitím globální paměti

U tohoto algoritmu bude opět jedno vlákno zpracovávat jeden pixel obrázku, ale k vypočítání výsledné barvy pixelu bude muset přistupovat k dalším sousedním pixelům, což znamená více přístupu do globální paměti a tím větší výpočetní čas. Kernelu budou kromě ukazatelů na obrázky a jejich velikosti navíc předány parametry definující velikost masky a celkový součet hodnot v masce. Průchod pixely pokrytými maskou je vidět ve výpisu kódu 9, kde jsou použity dva cykly `for`. Proměnná `R` v cyklech určuje poloměr masky, jinak řečeno, udává počet pixelů od středového pixelu masky až po její okraj. Pro lepší představu je toto znázorněno na obrázku 6, kde je na masce 5×5 červěně označen středový pixel a oranžově 2 pixely tvořící poloměr masky.



Obrázek 6: Maska 5×5 pixelů se znázorněním jejího poloměru

```
for(int y2 =-R; y2<=R; y2++) {
        for(int x2 =-R; x2<=R; x2++) {

            x3 = x+x2;
            y3 = y+y2;
            x3 = __max(0, x3);
            x3 = __min(x3, width-1);
            y3 = __max(0, y3);
```

```
            y3 = __min(y3, height-1);
            indexb = y3 * width + x3;

            finalx  += g_idata[ indexb ].x;
            finaly  += g_idata[ indexb ].y;
            finalz  += g_idata[ indexb ].z;
        }
    }
```

Výpis 9: Průchod pixely pokrytými maskou

Abychom maskou nezasahovali mimo obrázek, musí být pixely za hranicí obrázku nahrazeny pixely okraje. K tomu slouží proměnné `x3` a `y3`, do kterých jsou nejrpve vypočítány aktuální indexy zpracovávaného pixelu maskou, které tedy mohou být za hranicí obrázku a poté pomocí funkcí `__min` a `__max` jsou jejich indexy ořezány na 0 nebo na maximální index, který je určen velikostí obrázku. Dále je do proměnné `indexb` vypočítán index jednorozměrného pole vstupního obrázku, ze kterého se načte požadovaný pixel. Jas každé barvy v pixelu je přičten do proměnné `final`, která postupně, jak cykly prochází maskou, sčítá jasy pixelů. Hodnota této proměnné je pak podělena součtem vah pixelů v masce, což představuje proměnná `S`, čímž jsou jasy zprůměrovány a poté uloženy do výstupního obrázku. Toto je vidět ve výpisu kódu 10.

```
g_odata[index].x = __min(255, finalx  / S);
g_odata[index].y = __min(255, finaly  / S);
g_odata[index].z = __min(255, finalz  / S);
```

Výpis 10: Výpočet finálních jasů každé barvy

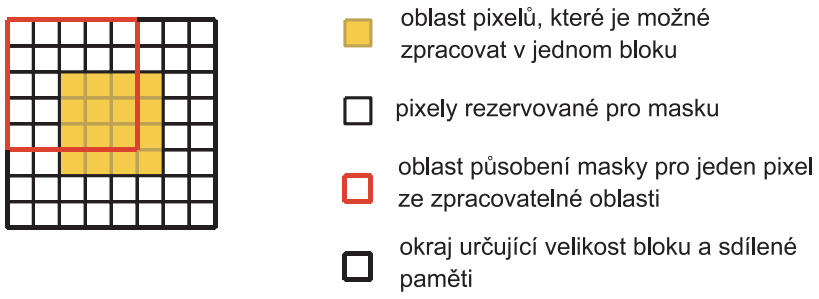
3.2.2 Rozmazání obrazu s použitím sdílené paměti

Použití sdílené paměti je poněkud komplikovanější, ale protože je přístupová doba do této paměti asi o dva desítkové řády menší, než do paměti globální, může nám její použití snížit výpočetní dobu algoritmu. Jak bylo zmíněno v kapitole 2, sdílená paměť je společná pouze pro vlákna jednoho bloku. Hlavním cílem je načíst pixely z bloku do jeho sdílené paměti a zpracovávat pixely až v ní. Rozměrová velikost sdílené paměti bude stačit stejná jako je velikost bloku. Každé vlákno bude mít k dispozici pouze pixely bloku, do kterého patří. Tímto je vymezena oblast, ve které může být použita maska. Velikost masky pak určuje velikost oblasti pixelů pro výsledný obraz, které je možné zpracovat v jednom bloku. Grafické znázornění tohoto systému je na obrázku 7.

Výsledný obraz nebude jako u předchozích algoritmů složen ze všech pixelů, ze kterých se skládají bloky, nýbrž pouze z části, která může být zpracována maskou, což znamená, že bude potřeba větší počet bloků na stejný obrázek než u předchozích algoritmů. Zbytek pixelů je v paměti pouze kvůli přesahu masky. Výpočet počtu bloků je vidět ve výpisu kódu 11, kde proměnné `TILE_W` a `TILE_H` značí šířku a výšku zpracovatelné oblasti pixelů v jednom bloku. Na obrázku 7 je tato oblast označena žlutou barvou.

```
dim3 blocks( ( sizex + TILE_W - 1 ) / TILE_W, ( sizey + TILE_H - 1 ) / TILE_H );
```

Výpis 11: Výpočet počtu bloků při použití sdílené paměti

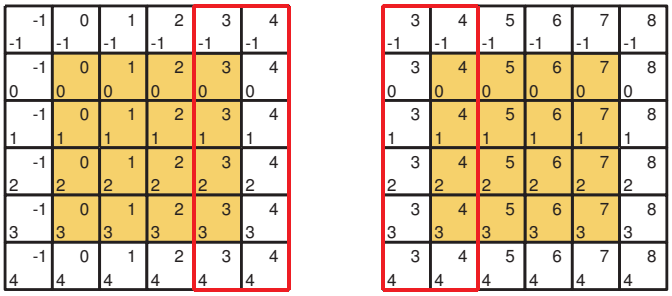


Obrázek 7: Příklad bloku se sdílenou pamětí a maskou

Na obrázku 8 lze vidět, jak na sebe jednotlivé bloky navazují. Aby na sebe bloky mohly takhle navazovat, je potřeba upravit indexy jednotlivých vláken. Toto se provede pomocí prvních dvou řádků ve výpisu kódu 12. Další řádky pak ořezávají přesahující indexy vláken na minimální a maximální možné indexy okrajů obrázku. Důvod je zřejmý z obrázku 8, kde lze vidět záporné indexy vláken.

```
int x = TILE_W * blockIdx.x + threadIdx.x - R;
int y = TILE_H * blockIdx.y + threadIdx.y - R;

x = __max(0, x);
x = __min(x, width-1);
y = __max(0, y);
y = __min(y, height-1);
```

Výpis 12: Výpočet indexů vláken pro práci se sdílenou pamětí



Obrázek 8: Ukázka indexů ve dvou blocích vláken a jejich navázání na sebe

Nyní má každé vlákno svůj správný index a může tak překopírovat pixel ze vstupního obrázku do sdílené paměti. K tomuto úkonu je ale ještě zapotřebí převést index vlákna na správný index pole vstupního obrázku a správný index pole sdílené paměti. Převody indexů a nakopírování pixelu ze vstupního obrázku do sdílené paměti lze vidět ve výpisu kódu 13.

```
unsigned int index = y * width + x;
```

```
unsigned int bindex = threadIdx.y * blockDim.y + threadIdx.x;

smem[bindex] = g_idata[index];
```

Výpis 13: Převod indexů vláken na indexy pole vstupního obrázku a pole sdílené paměti

Nyní se musí vlákna synchronizovat pomocí funkce `__syncthreads()`, což znamená, že na sebe vlákna v tomto bodě programu musí počkat a nesmí tak vykonávat další kód, dokud i to poslední vlákno nedokončí kód předchozí. Toto je velice důležité, neboť se v dalším kroku programu prochází pixely ve sdílené paměti, kde by mohly některé z pixelů chybět a vyvolat tak závažnou chybu programu.

Po synchronizaci vláken už potřebujeme pouze vlákna, která tvoří pixely pro výsledný obrázek. To jsou právě ta vlákna, která jsou na obrázku 8 označena žlutou barvou. Ostatní vlákna tvořící pomocnou oblast pro přesahující masku mohou být ukončena, protože již splnila svůj účel a nejsou k ničemu potřeba. K tomuto kroku je zapotřebí podmínka ve výpisu kódu 14, přes kterou projdou pouze potřebná vlákna, kde konstanty `BLOCK_W` a `BLOCK_H` určují velikost bloků.

```
if ((threadIdx.x >= R) && (threadIdx.x < (BLOCK_W−R)) && (threadIdx.y >= R) && (threadIdx.y <
    (BLOCK_H−R)))
```

Výpis 14: Podmínka pro filtraci již nepotřebných vláken

Vlákna, která projdou touto podmínkou, dále zpracovávají pixely pokryté maskou podobně jako u algoritmu s globální pamětí, jen s rozdílem přepočítání indexu, kde v tomto případě počítáme s indexy ze sdílené paměti. Cykly pro průchod maskou lze vidět ve výpisu kódu 15, kde proměnná `smem` je pole s pixely ve sdílené paměti.

```
for(int dy= −R; dy <= R; dy++) {
    for(int dx= −R; dx <= R; dx++) {
        indexc = bindex + (dy*blockDim.x) + dx;
        finalx  += smem[indexc].x;
        finaly  += smem[indexc].y;
        finalz  += smem[indexc].z;
    }
}
```

Výpis 15: Průchod pixely pokrytými maskou ve sdílené paměti

Nakonec se stejně jako u algoritmu využívajícího globální paměť podělí proměnné `final` počtem pixelů v masce a výsledný pixel se uloží do globální paměti. Toto lze vidět ve výpisu kódu 10.

3.3 Detekce hran

Hrany jsou místa v obraze, kde se prudce mění jasy pixelů. K nalezení takových míst slouží první a druhé derivace intenzit jasů. Jde o velikosti těchto derivací nebo změnu jejich znaménka. Největší hodnota derivace je ve směru kolmo na hranu. Ke zjednodušení se hrany detekují pouze ve dvou nebo čtyřech směrech. K detekci hran se používají

konvoluční masky, které hodnotu derivací aproximují. V této práci jsou použity 3 typy konvulučních masek.

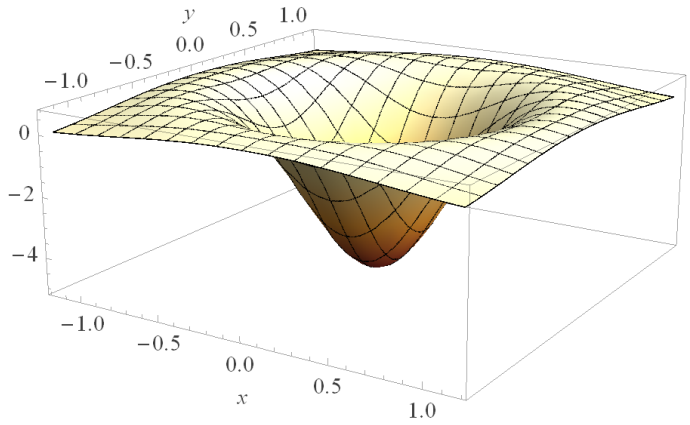
Jedná se o operátor Prewittové, který dokáže detekovat hrany pouze ve dvou směrech. Rotací masky, můžeme určovat kterými dvěma směry se mají hrany detekovat. Je založen na principu gradientu a první derivaci.

Laplaceův operátor detekuje hrany ve čtyřech směrech a je založen na druhé derivaci. Používá se k detekci hlavně izolovaných bodů, což může způsobovat šum. Lze ho použít ve dvou variantách, pozitivní a negativní. Po malé úpravě masky ho lze také využít k ostření obrazu. Tato operace se nazývá superpozice a prostřednímu pixelu masky v negativní formě je přičtena jednička, takže se původní obraz sečte s obrazem, kde jsou pouze hrany.

Laplacian of Gaussian je operace, která spojuje Laplaceův a Gaussův operátor. Při použití pouze Gaussova operátoru ke konvoluci obrazu by byl výsledný obraz rozmazaný, což je výhodné v tom, že použití Laplaceova operátoru ke konvoluci na takto upravený obraz způsobí méně šumu. Obě tyto operace lze spojit do jediné a dostat tak dvourozměrnou funkci definovanou následující rovnicí [10].

$$LoG(x,y) = \frac{(x^2+y^2)-2\sigma^2}{2\pi\sigma^6}\mathrm{e}^{-\frac{x^2+y^2}{2\sigma^2}} \tag{1}$$

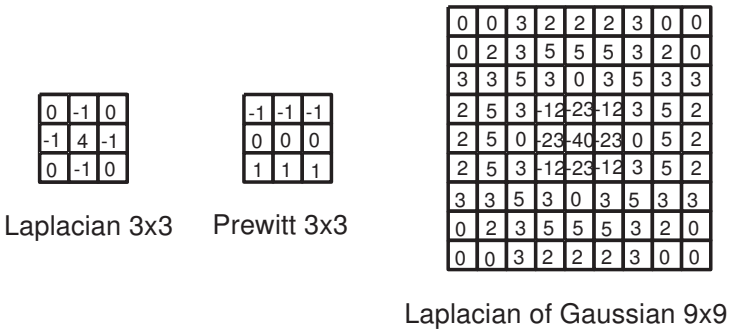
Na obrázku 9 je graf této funkce s parametrem $\sigma = 0.5$. Konvoluční maska, která aproximuje tuto funkci, je na obrázku 10, kde jsou také příklady předešlých masek.



Obrázek 9: Graf funkce Laplacian of Gaussian s parametrem $\sigma = 0.5$

3.3.1 CUDA program pro výpočet detekce hran

Algoritmus pro výpočet detekce hran vychází z předchozího algoritmu pro rozmazání obrazu, s tím rozdílem, že maska nyní neobsahuje pouze jedničky, ale je složena z různých číslic a tudíž musí být maska grafické kartě předána k výpočtu. Protože se maska za běhu kernelu nebude měnit, lze ji umístit do paměti konstant a tak zajistit, že data masky budou cachována. Alokace paměti probíhá před spuštěním kernelu pomocí funkce



Obrázek 10: Ukázky použitých masek

`cudaMemcpyToSymbol`, kde prvním parametrem je proměnná, v našem případě pole, které se nachází v paměti konstant, kam chceme pole s maskou nakopírovat. Definice proměnné pro paměť konstant musí obsahovat klíčové slovo `__constant__`. Aby k proměnným měl přístup samotný kernel, jsou definovány jako globální a nemusí se tak kernelu předávat. Celý algoritmus pak vypadá stejně jako algoritmus pro rozmazání obrazu s rozdílem v průchodu maskou. Ve výpisu kódu 16 lze vidět cykly pro průchod pixely pokrytými maskou, kde je každý z nich vynásoben právě tím číslem masky, pod kterým se nachází. Díky velkým a záporným hodnotám v maskách pro detekci hran je třeba do algoritmů přidat kontrolu hodnot jasů, které mohou být mimo svůj rozsah. Ve výpisu kódu 17 lze vidět ořezání těchto hodnot na minimum, či maximum.

```
int i = 0;
for(int dy= -R; dy <= R; dy++) {
   for(int dx= -R; dx <= R; dx++) {
      indexc = bindex + (dy*blockDim.x) + dx;
       finalx  += smem[indexc].x * mask[i];
       finaly  += smem[indexc].y * mask[i];
       finalz  += smem[indexc].z * mask[i];
      i++;
   }
}
```

Výpis 16: Průchod pixely pokrytými maskou a jejich úprava pomocí hodnot v masce

```
finalx = __max(0, finalx);
finaly = __max(0, finaly);
finalz = __max(0, finalz);

finalx = __min(255, finalx);
finaly = __min(255, finaly);
finalz = __min(255, finalz);
```

Výpis 17: Kontrola hodnot jasů a jejich případné ořezání na minimum či maximum

3.4 Bilineární interpolace

Pokud máme funkci, kde jsou známy hodnoty jen v některých bodech, použitím interpolace můžeme dostat přibližné hodnoty i v jiných bodech, které funkce vůbec neurčuje. Jednou z jednoduších je lineární interpolace, která spojuje dva sousední body přímkou, na které se pak nacházejí nově nalezené body. Polohy těchto nově nalezených bodů jsou tedy ovlivněny dvěma body, mezi kterými se nacházejí. Ve zpracování obrazu se používá bilineární interpolace, která pracuje se čtyřmi sousedními body a v této práci slouží ke zmenšení obrazu. Sousedními body jsou myšleny body s těmito souřadnicemi: $(i, j), (i + 1, j), (i, j + 1), (i + 1, j + 1)$. Poloha vypočtené bodu je pak $(i + \Delta i, j + \Delta j)$, kde Δi a Δj jsou neceločíselné vzdálenosti od bodu (i, j). Bilineární interpolace je pak dána následující rovnicí [10].

$$f(i + \Delta i, j + \Delta j) = a\Delta i + b\Delta j + c\Delta i\Delta j + d \tag{2}$$

Koeficienty a, b, c, d se určí, když za Δi a Δj dosadíme jedničky nebo nuly, tedy dosadíme relativní polohy čtyř sousedních bodů. Výsledné vztahy pro jednotlivé koeficienty vypadají následovně [10].

$$\begin{aligned} d &= f(i, j) && (3)\\ b &= f(i, j + 1) - f(i, j) && (4)\\ a &= f(i + 1, j) - f(i, j) && (5)\\ c &= f(i + 1, j + 1) + f(i, j) - f(i + 1, j) - f(i, j + 1) && (6) \end{aligned}$$

Po dosazení těchto vztahů do rovnice bilineární interpolace dostaneme rovnici pro výpočet hodnoty jasu z hodnot jasů čtyř sousedních bodů [10].

$$\begin{aligned} f(i + \Delta i, j + \Delta j) &= [f(i + 1, j) - f(i, j)]\Delta i + [f(i, j + 1) - f(i, j)]\Delta j + \\ &+ [f(i + 1, j + 1) + f(i, j) - f(i + 1, j) - f(i, j + 1)]\Delta i\Delta j + \\ &+ f(i, j) \end{aligned} \tag{7}$$

3.4.1 CUDA program pro výpočet bilineární interpolace

Před spuštěním kernelu je u tohoto algoritmu třeba spočítat velikost výsledného obrázku. Ve výpisu kódu 18 lze vidět předání parametru `percent` označujícího velikost nového obrázku v procentech a ve výpisu kódu 19 lze vidět výpočet nové velikosti obrázku.

```
void runInterpol( uchar4 *inputPic, uchar4* finalPic, int sizex, int sizey, int percent)
```

Výpis 18: Předávané parametry pro funkci bilineární interpolace

```
float scale = percent / (float)100;
int newSizex = (int)(sizex * scale);
int newSizey = (int)(sizey * scale);
```

Výpis 19: Výpočet nové velikosti obrázku

Nová velikost obrázku je potřeba k vytvoření správně veliké mřížky, protože právě v ní se bude tvořit upravený obrázek. Funkci kernelu je navíc předána nová velikost obrázku a měřítko. Po spuštění kernelu se opět ukončí vlákna, která svým indexem přesahují obrázek. Jak bylo zmíněno výše, je třeba vypočítat parametry bilineární rovnice a, b, c, d. Ve výpisu kódu 20 je výpočet parametru a pro každou barvu. Obdobně se pak vypočítají zbývající parametry.

```
b1.x = inputPic[ (int)((y + 0) / scale) * oldSizex + (int)((x + 0) / scale) ].x;
b1.y = inputPic[ (int)((y + 0) / scale) * oldSizex + (int)((x + 0) / scale) ].y;
b1.z = inputPic[ (int)((y + 0) / scale) * oldSizex + (int)((x + 0) / scale) ].z;
```

Výpis 20: Výpočet parametrů rovnice bilineární interpolace pro každou barvu

Další věcí co je potřeba výpočítat jsou vzdálenosti Δi a Δj. Jejich výpočet je ve výpisu kódu 21.

```
dx = (x / scale) - (int)(x / scale);
dy = (y / scale) - (int)(y / scale);
```

Výpis 21: Výpočet vzdáleností Δi a Δj

Nyní je možné spočítat výsledný jas nového pixelu. Ukázka výpočtu je ve výpisu kódu 22, kde jsou vypočítány výsledné jasy pro všechny tři barvy.

```
result_b.x = dx * (b2.x - b1.x) + dy * (b3.x - b1.x) + dx * dy * (b4.x + b1.x - b2.x - b3.x) + b1.
    x
result_b.y = dx * (b2.y - b1.y) + dy * (b3.y - b1.y) + dx * dy * (b4.y + b1.y - b2.y - b3.y) + b1.
    y;
result_b.z = dx * (b2.z - b1.z) + dy * (b3.z - b1.z) + dx * dy * (b4.z + b1.z - b2.z - b3.z) + b1.
    z;
```

Výpis 22: Výpočet bilineární interpolace

Nakonec je opět potřeba přetypovat a ořezat výsledné jasy do našeho intervalu pomocí funkcí `__min()` a `__max()`.

4 Výsledky měření

Měření bylo provedeno na běžném počítači konkrétně na domácí PC sestavě se čtyřjádrovým procesorem AMD Athlon II X4 640 s frekvencí 3,0 GHz a grafické kartě NVIDIA GeForce GTX 650 s taktem jádra 1072 MHz a 384 CUDA jádry. Výsledky všech měření lze vidět v příloze B v tabulkách 1 - 6.

Zpracování grafickou kartou bylo u všech algoritmů a s různými velikostmi obrázků rychlejší než zpracování pomocí CPU kromě algoritmu bilineární interpolace (tabulka 6), kde byl u velmi malého obrázku CPU rychlejší. Se zvětšující se velikostí obrázků rostly rozdíly v rychlosti zpracování, kde u algoritmu rozmazání s maskou 17×17 (tabulka 3) dosáhla grafická karta přibližně $200\times$ nižšího času. Efektivita zpracování pomocí NVIDIA CUDA tedy roste s počtem vyvolaných vláken.

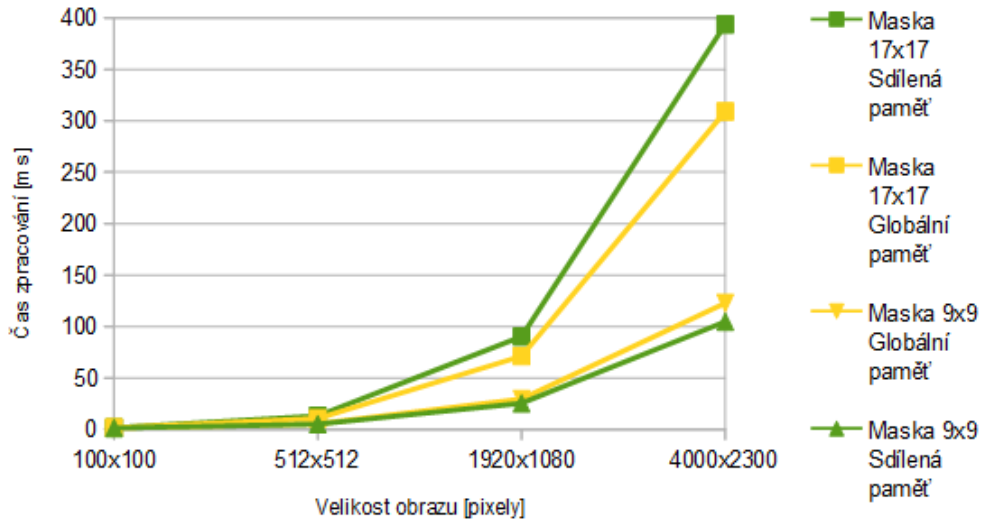
Další věcí, která velmi ovlivňuje výsledný čas je to, že se musí nejprve celý obraz překopírovat do karty před samotným zpracováním, což je operace velice drahá na čas. Také proto je u nejmenšího měřeného obrázku 100×100 pixelů rozdíl v čase zpracování mezi GPU a CPU nejmenší.

Rozdíl mezi zpracováním obrazu s pomocí sdílené paměti a s pomocí pouze globální není tolik markantní. Největší rozdíl byl při použití masky velkých rozměrů, kde algoritmus se sdílenou pamětí postupně ztrácel s přibývající velikostí obrazu, což lze vidět v tabulce 3. U algoritmu detekce hran pomocí funkce Laplacian of Gaussian byly časy s použitím sdílené paměti naopak lepší než časy s použitím pouze globální paměti (tabulka 5). Tedy při vhodné velikosti obrázku a vhodně velké masce lze se sdílenou pamětí dosáhnout lepších výsledků.

I když je sdílená paměť velmi rychlá, je třeba myslet na to, že abychom s ní mohli pracovat, musíme přece jen přistoupit do globální paměti a vykopírovat z ní pixely do paměti sdílené. Při použití algoritmů s maskou navíc kvůli jejímu přesahu některé pixely kopírujeme vícekrát. Čím větší je maska, tím se více pixelů kopíruje navíc. Na obrázku 11 lze vidět graf, který ukazuje rozdíly v časech zpracování u algoritmu používajícího pouze globální paměť a algoritmu používajícího i sdílenou paměť při použití masek dvou různých velikostí a různých velikostí zpracovávaného obrazu.

U algoritmu bilineární interpolace záleží rychlost zpracování na velikosti výsledného obrázku. Čím je zmenšení větší a obrázek menší, tím je potřeba zpracovat méně pixelů. Jak bylo zmíněno výše, efektivita zpracování pomocí GPU narůstá s potřebou zpracování většího množství pixelů, což lze vidět v tabulce 6. Také v této tabulce lze vidět, že při velmi malém obrázku a jeho velkém zmenšení může být CPU rychlejší než GPU. Je to z důvodu velké režie při kopírování dat z a do grafické karty.

Poznámka 4.1 Procentuální zrychlení je počítáno z času výpočtu CPU a z času GPU. Protože jsou časy GPU u některých algoritmů dva, byl zvolen ten rychlejší.



Obrázek 11: Graf znázorňující rozdíl v časech mezi použitím sdílené paměti a pouze globální pamětí u algoritmů konvoluce pracujících s konvolučnimi maskami.

5 Závěr

Grafické karty byly předně používány k akceleraci 2D a 3D obrazů her a čím se zvyšovaly nároky na realističnost obrazu, tím více se karty vyvíjely a zdokonalovaly. Použití karet k jiným paralelním výpočtům bylo kvůli složitosti velice obtížné a moc se nevyužívalo. Společnost NVIDIA přinesla na trh novou architekturu grafických čipů použitelnou i k jiným výpočetním úlohám a bylo vyvinuto prostředí CUDA, ve kterém je možné programovat grafické aplikace v jazyce C [11]. V této práci jsem použil technologii CUDA k vytvoření aplikace, která slouží k masivnímu paralelní zpracování obrazu. Z výsledků práce je výhoda použití grafických čipů v paralelních výpočtech viditelná. Použití technologie CUDA je však omezeno pouze na grafické karty společnosti NVIDIA. Stejně je tomu i u konkurenční společnosti AMD, kde lze jejich technologii používat výhradně na jejich grafických kartách. Protože jsou tyto koncepty dělány na míru architekturám grafických karet, lze očekávat i dosažení lepších výpočetních rychlostí než u standardu OpenCL, který lze však použít na karty obou společností.

V dnešní době už existuje spousta aplikací využívajících k výpočtům grafické karty a můžeme očekávat, že tento trend bude díky skvělým výpočetním časům pokračovat. Pokud budou podmínky pro vytváření aplikací využívajících grafické karty stále lepší, je možné že takové aplikace budou v budoucnosti naprosto běžné. Možná to bude díky fyzikálním omezením výroby čipů nevyhnutelné. Navíc programování paralelních aplikací už není tolik složité a věřím, že se bude dále zjednodušovat. Už teď se dá vyvíjet CUDA program pod více známými programovacími jazyky jako jsou C, C++, C#, Java, Fortran, Python a další. Osobně bych hodnotil programování v NVIDIA CUDA jako programátorsky velmi přívětivé. Od programátora se v základu vyžaduje pochopení běhu CUDA programu, znalost uspořádání vláken ve vytvořené mřížce a porozumění komunikace s různými pamětmi grafické karty. Některé algoritmy ale mohou být náročnější na programátorovu představivost. V souhrnu je programování CUDA programu jednoduché, dokud není potřeba výrazné zefektivnění kódu. Pak si CUDA vyžaduje znát hlubší znalosti a více souvislostí.

Úkolem této bakalářské práce bylo vytvořit program, který bude umět otestovat algoritmy pro zpracování obrazu a ve vhodné formě prezentovat jejich výsledky, což jsem splnil. Za pomoci vývojového prostředí QT Creator jsem vytvořil grafické uživatelské rozhraní s možností otevření libovolného obrázku a jeho úpravy pomocí GPU a CPU. Při zpracování pomocí GPU je možnost vybrat algoritmus používající sdílenou nebo jen globální paměť. Výstupem programu je náhled na zpracovaný obrázek a čas samotného zpracování. Náhledy na vytvořený program pro účely této práce lze vidět v příloze C na obrázcích 12 - 16.

6 Reference

[1] KADLEC, Václav. *Učíme se programovat v jazyce C.* Vyd. 1. Praha: Computer Press, 2002, 277 s. ISBN 80-722-6715-9.

[2] BLANCHETTE, Jasmin a Mark SUMMERFIELD. *C GUI programming with Qt 4.* Upper Saddle River, NJ: Pearson Hall in association with Trolltech Press, c2006, xvii, 537 p. ISBN 978-013-1872-493.

[3] OpenCV. *Documentation* [online]. [cit. 2014-04-21]. last revision 21th of April 2014 Dostupné z: <http://docs.opencv.org>.

[4] YLILUOMA, Joel. *Guide into OpenMP: Easy multithreading programming for C++* [online]. [cit. 2014-04-21]. last December 2008 Dostupné z: <http://bisqwit.iki.fi/story/howto/openmp/>.

[5] NVIDIA. *ABOUT CUDA* [online]. [cit. 2014-05-04]. Dostupné z: <https://developer.nvidia.com/about-cuda>.

[6] NVIDIA. *CUDA C Programming Guide* [online]. v6.0, last revision 13th of February 2014 [cit. 2014-04-21]. Dostupné z: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>.

[7] NVIDIA. *CUDA C Best Practices Guide* [online]. v6.0, last revision 13th of February 2014 [cit. 2014-04-21]. Dostupné z: <http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/>.

[8] FARBER, Rob. *CUDA, Supercomputing for the Masses: Part 11* [online]. Last revision 18th of March 2009 [cit. 2014-04-21]. Dostupné z: <http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/215900921>.

[9] CMLAB.CSIE.NTU.EDU.TW. *NVIDIA CUDA Library Documentation* [online]. v4.2, last revision 31th of May 2012 [cit. 2014-04-21]. Dostupné z: <http://graphics.im.ntu.edu.tw/ bossliaw/nvCuda_doxygen/html/index.html>.

[10] DOBEŠ, Michal. *Zpracování obrazu a algoritmy v C#.* 1. vyd. Praha: BEN - technická literatura, 2008, 143 s. ISBN 978-80-7300-233-6.

[11] VÉLE, Adam. *GPU computing – aneb grafické karty útočí* [online]. [cit. 2014-05-02]. Dostupné z: <http://www.cad.cz/hardware/78-hardware/2106-gpu-computing-aneb-graficke-karty-utoci.html>.

Seznam příloh

A Příloha na CD

Kód celého programu vytvořeného pro tuto práci se nachází na přiloženém CD v adresáři `Program`.

B Tabulky s časy měření

Velikost obrázku [pixely]	GPU [ms]	CPU [ms]	Zrychlení [%]
100×100	1,09	6,00	453
512×512	3,70	155,94	4112
1920×1080	16,33	1241,06	7499
4000×2300	66,09	5488,56	8204

Tabulka 1: Výsledky gamma korekce

Velikost obrázku [pixely]	GPU se sdílenou pamětí [ms]	GPU jen s globální pamětí [ms]	CPU [ms]	Zrychlení [%]
100×100	1,07	1,05	2,72	159
512×512	3,31	3,27	73,39	2143
1920×1080	14,15	14,81	573,68	3955
4000×2300	55,84	55,71	2465,34	4325

Tabulka 2: Výsledky rozmazání s maskou 3×3

Velikost obrázku [pixely]	GPU se sdílenou pamětí [ms]	GPU jen s globální pamětí [ms]	CPU [ms]	Zrychlení [%]
100×100	1,57	1,51	65,50	4245
512×512	12,98	10,55	1849,01	17 419
1920×1080	90,18	71,05	14 302,76	20 031
4000×2300	393,17	309,02	62 334,57	20 072

Tabulka 3: Výsledky rozmazání s maskou 17×17

Velikost obrázku [pixely]	GPU se sdílenou pamětí [ms]	GPU jen s globální pamětí [ms]	CPU [ms]	Zrychlení [%]
100×100	1,13	1,14	2,54	122
512×512	3,33	3,31	64,58	1853
1920×1080	13,81	13,18	511,50	3781
4000×2300	54,31	51,96	2281,11	4290

Tabulka 4: Výsledky detekce hran pomocí Laplaceova operátoru s maskou 3×3

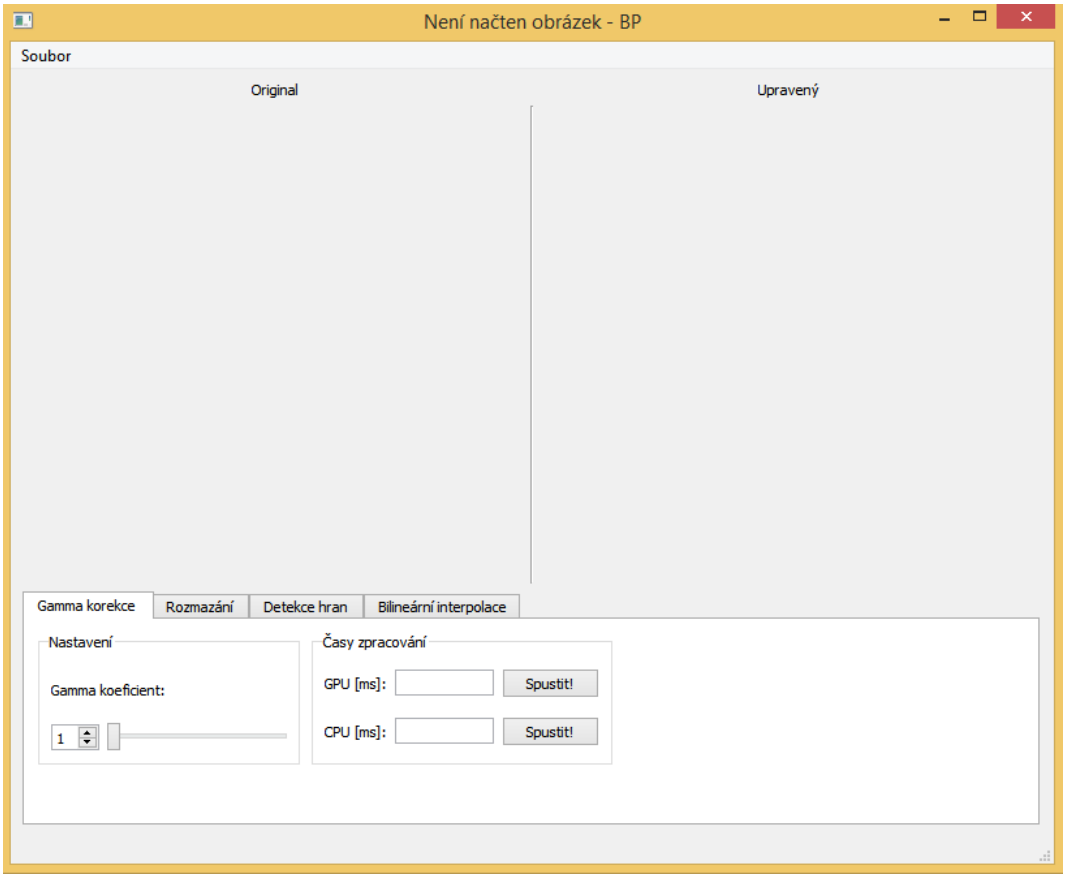
Velikost obrázku [pixely]	GPU se sdílenou pamětí [ms]	GPU jen s globální pamětí [ms]	CPU [ms]	Zrychlení [%]
100×100	1,20	1,27	20,57	1615
512×512	5,02	5,41	526,33	10 382
1920×1080	25,30	29,48	4097,80	16 096
4000×2300	104,71	122,72	18 796,77	17 851

Tabulka 5: Výsledky detekce hran pomocí funkce Laplacian of Gaussian s maskou 9×9

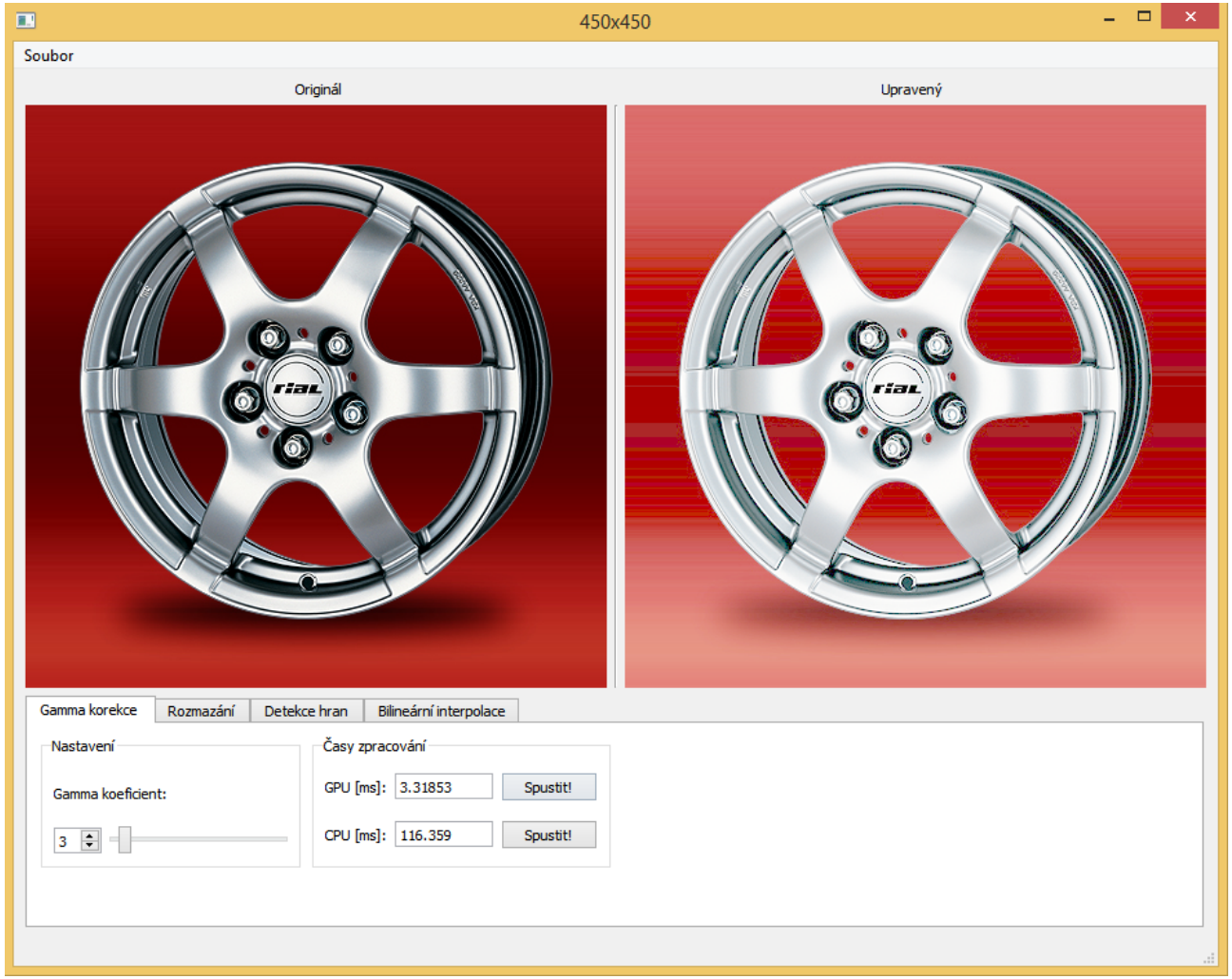
Velikost obrázku [pixely]	Zmenšení na [%]	GPU [ms]	CPU [ms]	Zrychlení [%]
100×100	15	1,74	0,08	2071
100×100	65	1,12	1,41	26
100×100	99	1,08	3,28	203
1600×1200	15	6,11	14,34	135
1600×1200	65	8,74	269,30	2983
1600×1200	99	12,36	622,70	4939

Tabulka 6: Výsledky bilineární interpolace

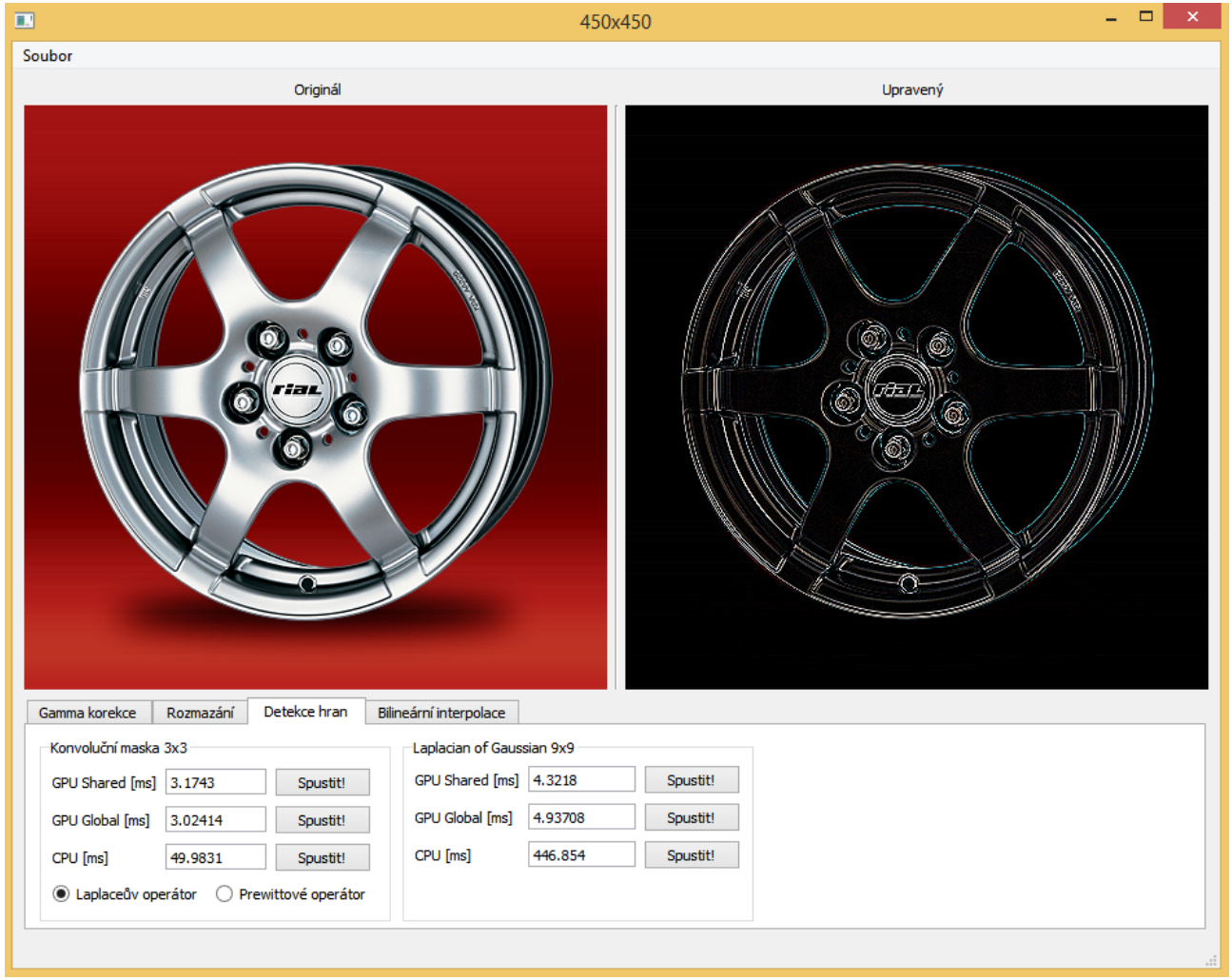
C Screenshoty programu



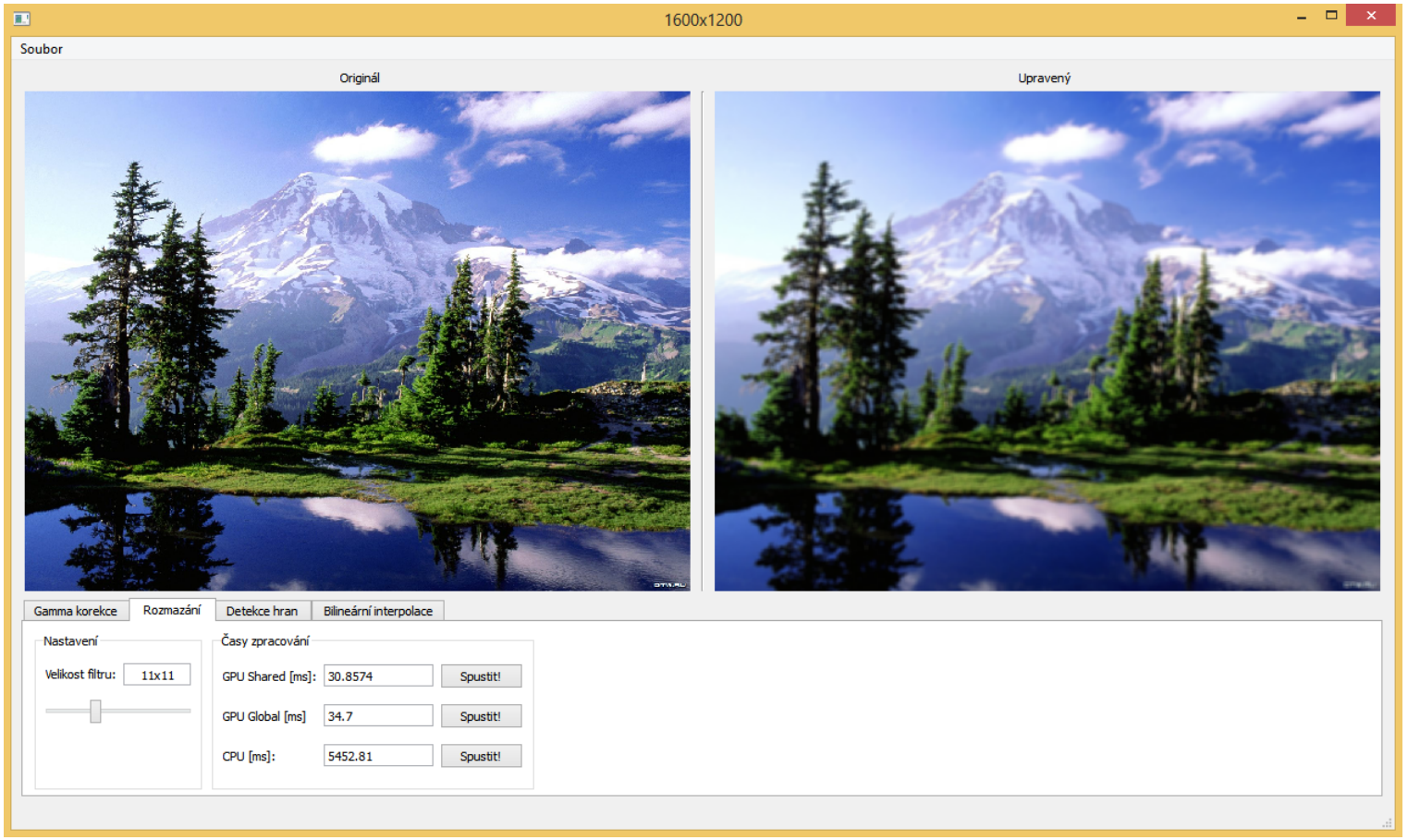
Obrázek 12: Náhled na vytvořený program ihned po jeho spuštění



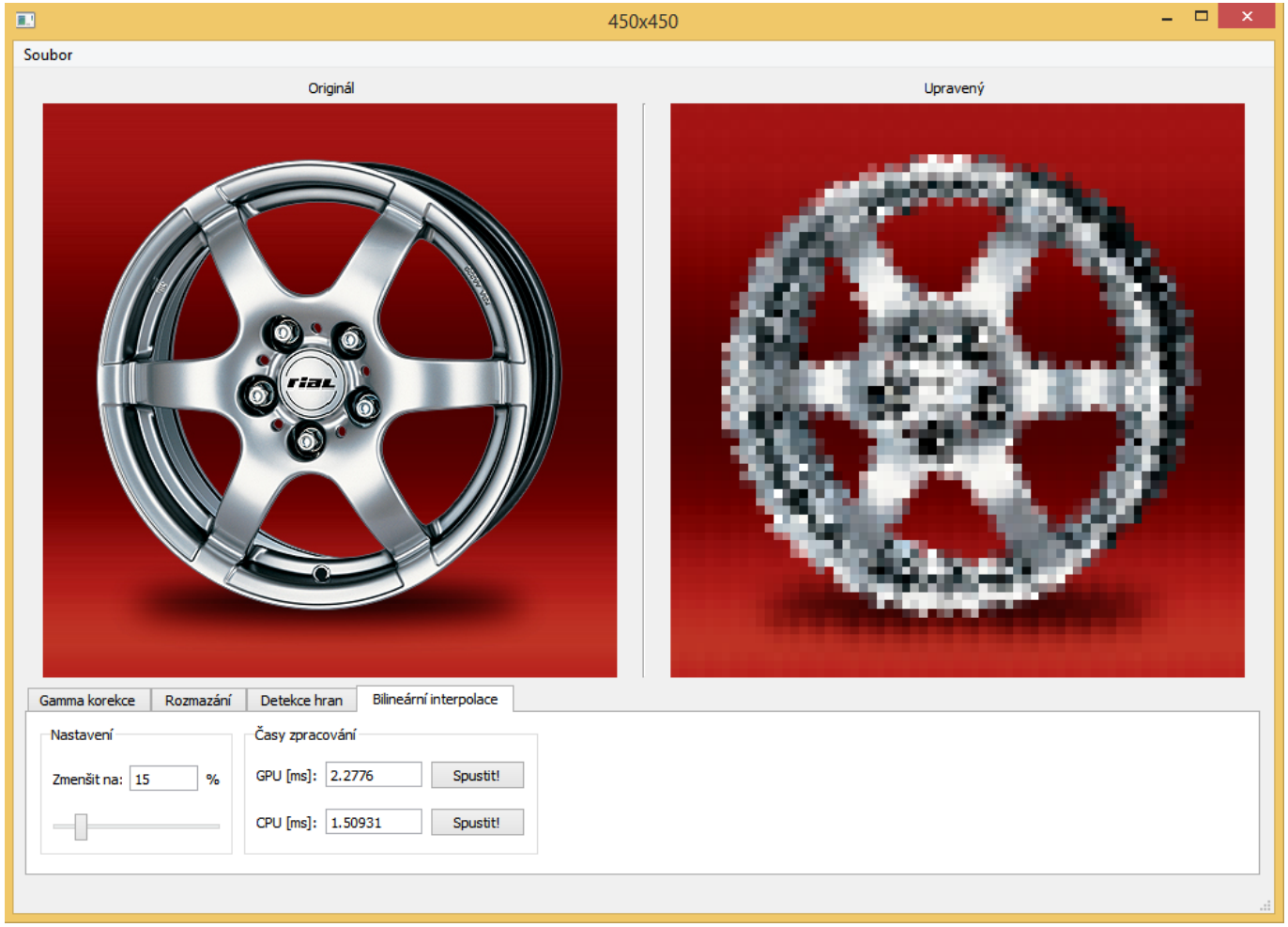
Obrázek 13: Gamma korekce



Obrázek 14: Detekce hran



Obrázek 15: Rozmazání



Obrázek 16: Bilineární interpolace (výsledný obrázek je zvětšený, protože se přizpůsobuje velikosti okna, proto je vidět „rozpixelovaně“)